**Experiment no – 04**

**Aim:** Write a program to construct NFA using given regular expression.

**Algorithm:**

1.  Create a menu for getting four regular expressions input as choice.

2.   To draw NFA for a, a/b ,ab ,a\* create a routine for each regular expression.

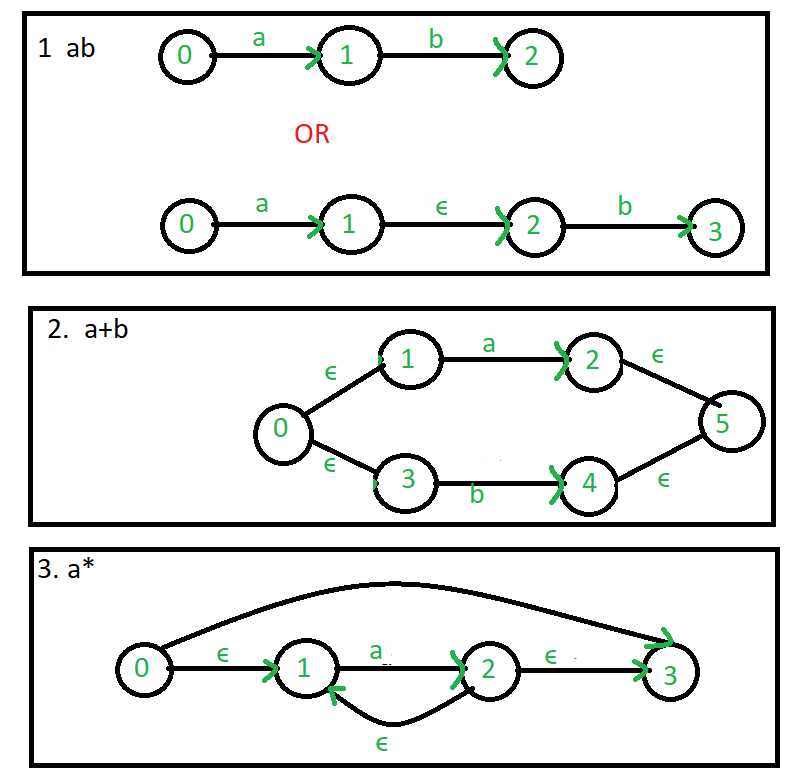
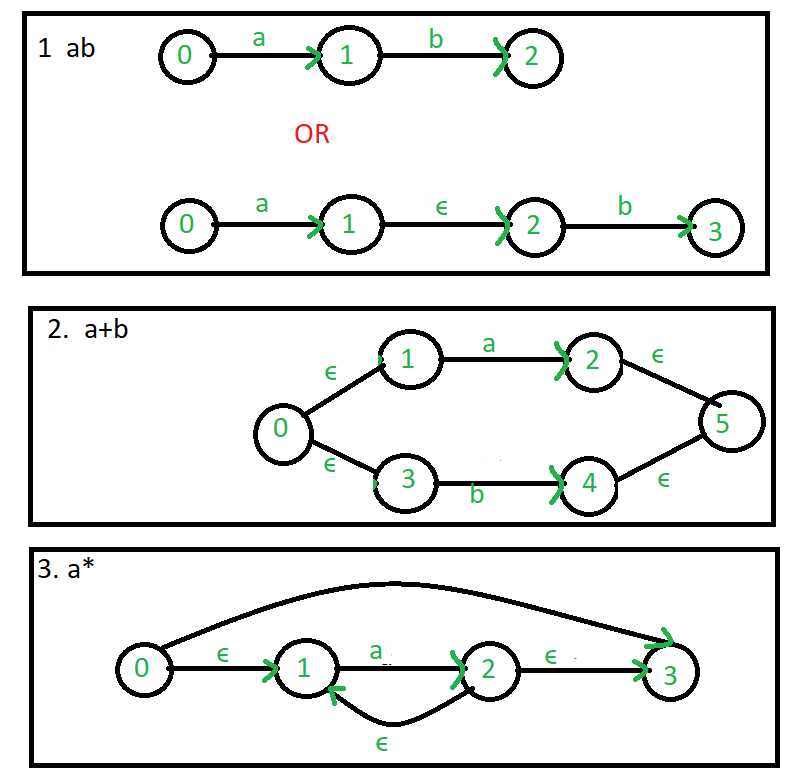
3.  For converting from regular expression to NFA, certain transition had been made based on choice of input at the rumtime.

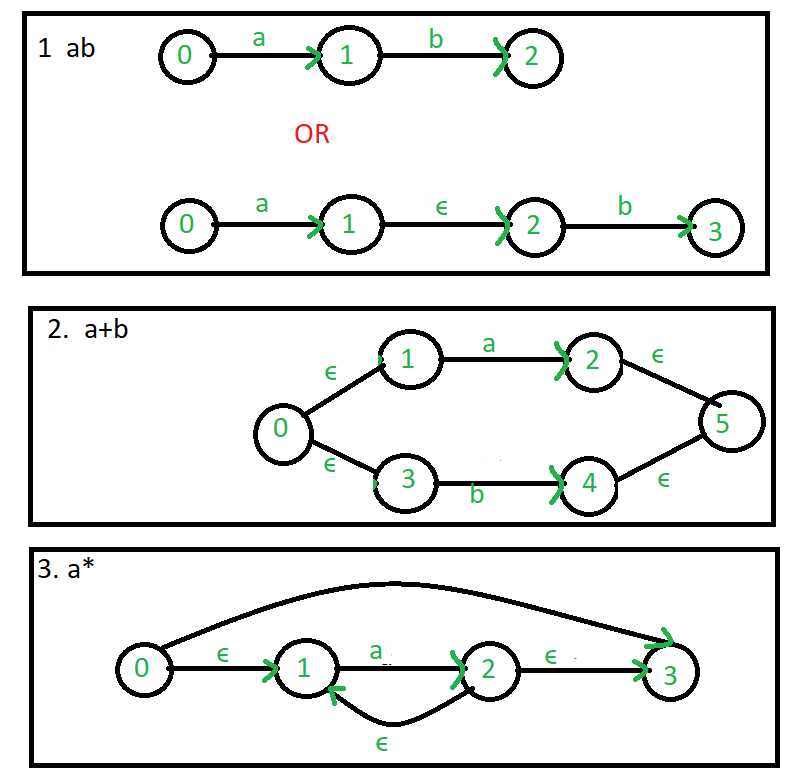
4.  Each of the NFA will be displayed is sequential order.

**Theory Explanation :**

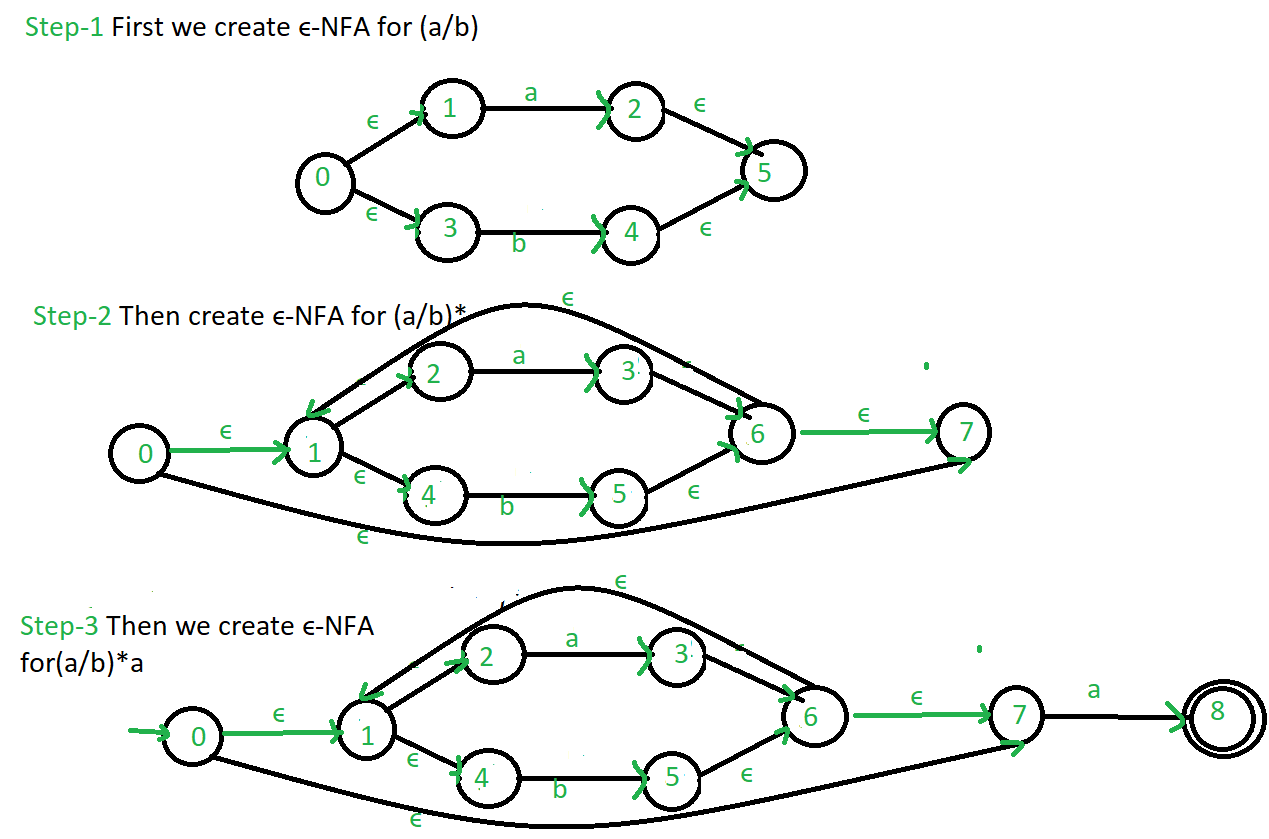
∈-NFA is similar to the NFA but have minor difference by epsilon move. This automaton replaces the transition function with the one that allows the empty string ∈ as a possible input. The transitions without consuming an input symbol are called ∈-transitions.

In the state diagrams, they are usually labeled with the Greek letter ∈. ∈-transitions provide a convenient way of modeling the systems whose current states are not precisely known: i.e., if we are modeling a system and it is not clear whether the current state (after processing some input string) should be q or q’, then we can add an ∈-transition between these two states, thus putting the automaton in both states simultaneously.

Common regular expression used in make ∈-NFA: 



Example: Create a ∈-NFA for regular expression: (a/b)\*a



**Program:**

1. **Main.Py**

***#Program to Construct NFA using REGEX***

**import** sys  
**import** nfa\_utils  
**import** time  
  
*# print the intro text block***with** open(**"intro.dat"**) **as** intro\_file:  
 print(intro\_file.read())  
  
*# regular expression string to compare against provided input*regex = **None**regex\_nfa = **None***# last line of user input read from the command line*line\_read = **""***# continuously parse and process user input***while True**:  
 *# read in line of user input* line\_read = input(**"> "**)  
 *# make a lowercase copy of the input for case insensitive comparisons* line\_read\_lower = line\_read.lower()  
  
 **if** line\_read\_lower == **"exit"**:  
 *# exit the program* print(**"\nExiting..."**)  
 sys.exit()  
  
 **if** line\_read\_lower.startswith(**"regex="**):  
 *# user wants to set the regex to a string they've provided* regex = line\_read[6:]  
 print(**"New regex pattern:"**, regex, **"\n"**)  
 start\_time = time.time()  
 *# turn regular expression string into an NFA object* regex\_nfa = nfa\_utils.get\_regex\_nfa(regex)  
 regex\_nfa.reset()  
 finish\_time = time.time()  
 ms\_taken = (finish\_time - start\_time) \* 1000  
  
 print(**"\nBuilt NFA in {:.3f} ms.\n"**.format(ms\_taken))  
 print(regex\_nfa)  
 **else**:  
 *# assume the user intends to test this entered string against the regex* **if** regex\_nfa **is None**:  
 *# regex has not yet been set* print(**"Please supply a regular expression string first, with regex=(regex here)"**)  
 **else**:  
 start\_time = time.time()  
 *# feed input string into NFA* regex\_nfa.feed\_symbols(line\_read, return\_if\_dies=**True**)  
 accepts = regex\_nfa.is\_accepting()  
 finish\_time = time.time()  
 ms\_taken = (finish\_time - start\_time) \* 1000  
  
 print(**"String was {} by NFA"** .format(**"ACCEPTED" if** accepts **else "REJECTED"**))  
  
 print(**"Calculated in {:.3f} ms."**.format(ms\_taken))  
  
 *# print(regex\_nfa)* regex\_nfa.reset()  
  
 *# print a new line for aesthetics* print()

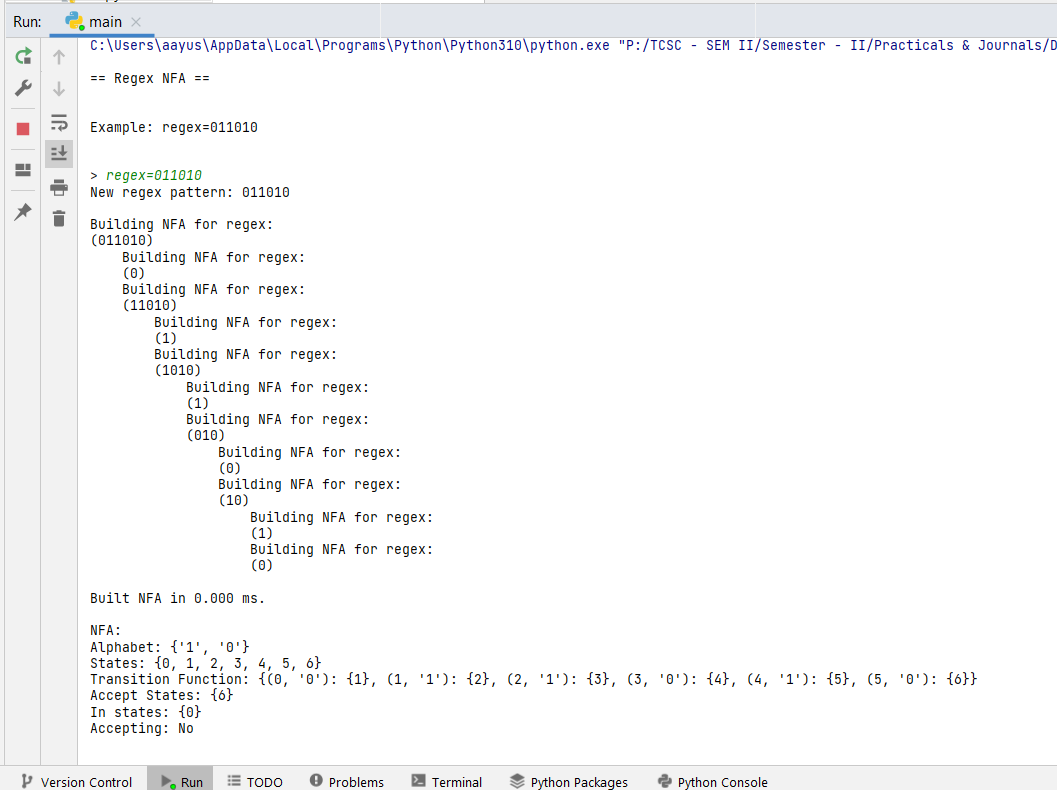
1. **NFA.py**

**class** NFA:  
 *"""Class representing a non-deterministic finite automaton"""* **def** \_\_init\_\_(self):  
 *"""Creates a blank NFA"""  
  
 # all NFAs have a single initial state by default* self.alphabet = set()  
 self.states = {0}  
 self.transition\_function = {}  
 self.accept\_states = set()  
  
 *# set of states that the NFA is currently in* self.in\_states = {0}  
  
 **def** add\_state(self, state, accepts=**False**):  
 self.states.add(state)  
  
 **if** accepts:  
 self.accept\_states.add(state)  
  
 **def** add\_transition(self, from\_state, symbol, to\_states):  
 self.transition\_function[(from\_state, symbol)] = to\_states  
  
 **if** symbol != **""**:  
 self.alphabet.add(symbol)  
  
 **def** feed\_symbol(self, symbol):  
 *"""  
 Feeds a symbol into the NFA, calculating which states the  
 NFA is now in, based on which states it used to be in  
 """  
  
 # a dead NFA will not have any transitions after a symbol is fed in* **if** self.is\_dead():  
 **return** new\_states = set()  
  
 *# process each old state in turn* **for** state **in** self.in\_states:  
 pair = (state, symbol)  
  
 *# check for a legal transition from the old state to a  
 # new state, based on what symbol was fed in* **if** pair **in** self.transition\_function:  
 *# add the corresponding new state to the updated states list* new\_states |= self.transition\_function[pair]  
  
 self.in\_states = new\_states  
  
 *# feed the empty string through the nfa* self.feed\_empty()  
  
 **def** feed\_symbols(self, symbols, return\_if\_dies=**False**):  
 *"""  
 Feeds an iterable into the NFAs feed\_symbol method  
  
 :param symbols: Iterable of symbols to feed through the NFA  
 :param return\_if\_dies: If true, ignore any further symbols after the NFA dies (for efficiency),  
 since a dead NFA will never accept, regardless of any further input.  
 """* **for** symbol **in** symbols:  
 self.feed\_symbol(symbol)  
  
 **if** return\_if\_dies **and** self.is\_dead():  
 *# NFA is dead; feeding further symbols will not change the NFA's state* **return  
  
 def** feed\_empty(self):  
 *"""  
 Continuously feeds empty strings into the NFA until they fail  
 to cause any further state transitions  
 """  
  
 # a dead NFA will not have any empty string transitions* **if** self.is\_dead():  
 **return** old\_states\_len = **None** *# set of states that will be fed the empty string on the next pass* unproc\_states = self.in\_states  
 first\_run = **True** *# keep feeding the empty string until no more new states are transitioned into* **while** first\_run **or** len(self.in\_states) > old\_states\_len:  
 old\_states\_len = len(self.in\_states)  
 *# set of new states transitioned into after the empty string was fed* new\_states = set()  
  
 *# process each state in turn* **for** state **in** unproc\_states:  
 pair = (state, **""**)  
  
 *# check if this state has a transition using the empty string  
 # to another state* **if** pair **in** self.transition\_function:  
 *# add the new state to a set to be added to self.in\_states later* new\_states |= self.transition\_function[pair]  
  
 *# merge new states back into "in" states* self.in\_states |= new\_states  
 *# all new states discovered will be fed the empty string on the next pass* unproc\_states = new\_states  
 first\_run = **False  
  
 def** is\_accepting(self):  
 *# accepts if we are in ANY accept states  
 # ie. if in\_states and accept\_states share any states in common* **return** len(self.in\_states & self.accept\_states) > 0  
  
 **def** is\_dead(self):  
 *"""  
 Returns true if the NFA is not in ANY states.  
 A "dead" NFA can never be in any states again.  
 """* **return** len(self.in\_states) == 0  
  
 **def** reset(self):  
 *"""  
 Resets the NFA by putting it back to it's initial state,  
 and feeding the empty string through it  
 """* self.in\_states = {0}  
 self.feed\_empty()  
  
 **def** \_\_str\_\_(self):  
 *"""  
 String representation of this NFA.  
 Useful for debugging.  
 """* **return "NFA:\n"** \  
 **"Alphabet: {}\n"** \  
 **"States: {}\n"** \  
 **"Transition Function: {}\n"** \  
 **"Accept States: {}\n"** \  
 **"In states: {}\n"** \  
 **"Accepting: {}\n"**\  
 .format(self.alphabet,  
 self.states,  
 self.transition\_function,  
 self.accept\_states,  
 self.in\_states,  
 **"Yes" if** self.is\_accepting() **else "No"**)  
  
 **def** \_\_eq\_\_(self, other):  
 *"""  
 Checks if two NFAs are equal. Used for testing.  
  
 Tests if they are structurally the same; does NOT check if they are in the same states.  
  
 Also ignores alphabets.  
 """* **return** self.states == other.states \  
 **and** self.transition\_function == other.transition\_function \  
 **and** self.accept\_states == other.accept\_states

1. **NFA.UTILS.py**

**from** nfa **import** NFA  
**import** copy  
  
  
**def** get\_single\_symbol\_regex(symbol):  
 *""" Returns an NFA that recognizes a single symbol """* nfa = NFA()  
 nfa.add\_state(1, **True**)  
 nfa.add\_transition(0, symbol, {1})  
  
 **return** nfa  
  
  
**def** shift(nfa, inc):  
 *"""  
 Increases the value of all states (including accept states and transition function etc)  
 of a given NFA bya given value.  
  
 This is useful for merging NFAs, to prevent overlapping states  
 """  
 # update NFA states* new\_states = set()  
 **for** state **in** nfa.states:  
 new\_states.add(state + inc)  
 nfa.states = new\_states  
  
 *# update NFA accept states* new\_accept\_states = set()  
 **for** state **in** nfa.accept\_states:  
 new\_accept\_states.add(state + inc)  
 nfa.accept\_states = new\_accept\_states  
  
 *# update NFA transition function* new\_transition\_function = {}  
 **for** pair **in** nfa.transition\_function:  
 to\_set = nfa.transition\_function[pair]  
 new\_to\_set = set()  
  
 **for** state **in** to\_set:  
 new\_to\_set.add(state + inc)  
  
 new\_key = (pair[0] + inc, pair[1])  
 new\_transition\_function[new\_key] = new\_to\_set  
  
 nfa.transition\_function = new\_transition\_function  
  
  
**def** merge(a, b):  
 *"""Merges two NFAs into one by combining their states and transition function"""* a.accept\_states = b.accept\_states  
 a.states |= b.states  
 a.transition\_function.update(b.transition\_function)  
 a.alphabet |= b.alphabet  
  
  
**def** get\_concat(a, b):  
 *""" Concatenates two NFAs, ie. the dot operator """  
  
 # number to add to each b state number  
 # this is to ensure each NFA has separate number ranges for their states  
 # one state overlaps; this is the state that connects a and b* add = max(a.states)  
  
 *# shift b's state/accept states/transition function, etc.* shift(b, add)  
  
 *# merge b into a* merge(a, b)  
  
 **return** a  
  
  
**def** get\_union(a, b):  
 *"""Returns the resulting union of two NFAs (the '|' operator)"""  
  
 # create a base NFA for the union* nfa = NFA()  
  
 *# clear a and b's accept states* a.accept\_states = set()  
 b.accept\_states = set()  
  
 *# merge a into the overall NFA* shift(a, 1)  
 merge(nfa, a)  
  
 *# merge b into the overall NFA* shift(b, max(nfa.states) + 1)  
 merge(nfa, b)  
  
 *# add an empty string transition from the initial state to the start of a and b  
 # (so that the NFA starts in the start of a and b at the same time)* nfa.add\_transition(0, **""**, {1, min(b.states)})  
  
 *# add an accept state at the end so if either a or b runs through,  
 # this NFA accepts* new\_accept = max(nfa.states) + 1  
 nfa.add\_state(new\_accept, **True**)  
 nfa.add\_transition(max(a.states), **""**, {new\_accept})  
 nfa.add\_transition(max(b.states), **""**, {new\_accept})  
  
 **return** nfa  
  
  
**def** get\_kleene\_star\_nfa(nfa):  
 *"""  
 Wraps an NFA inside a kleene star expression  
 (NFA passed in recognizes 0, 1 or many of the strings it originally recognized)  
 """  
 # clear old accept state* nfa.accept\_states = {}  
  
 *# shift NFA by 1 and insert new initial state* shift(nfa, 1)  
 nfa.add\_state(0)  
  
 *# add new ending accept state* last\_state = max(nfa.states)  
 new\_accept = last\_state + 1  
 nfa.add\_state(new\_accept, **True**)  
 nfa.add\_transition(last\_state, **""**, {new\_accept})  
  
 *# add remaining empty string transitions* nfa.add\_transition(0, **""**, {1, new\_accept})  
 nfa.add\_transition(last\_state, **""**, {0})  
  
 **return** nfa  
  
**def** get\_one\_or\_more\_of\_nfa(nfa):  
 *"""  
 Wraps an NFA inside the "one or more of" operator (plus symbol)  
  
 Simply combines the concatenation operator and the kleene star operator.  
 """  
  
 # must make a copy of the nfa,  
 # these functions operate on the nfa passed in, they do not make a copy* **return** get\_concat(copy.deepcopy(nfa), get\_kleene\_star\_nfa(nfa))  
  
**def** get\_zero\_or\_one\_of\_nfa(nfa):  
 *"""  
 Wraps an NFA inside the "zero or one of" operator (question mark symbol)  
  
 Simply uses the union operator, with one path for the empty string, and the other path  
 for the NFA being wrapped.  
 """* **return** get\_union(get\_single\_symbol\_regex(**""**), nfa)  
  
**def** get\_regex\_nfa(regex, indent=**""**):  
 *"""Recursively builds an NFA based on the given regex string"""* print(**"{0}Building NFA for regex:\n{0}({1})"**.format(indent, regex))  
 indent += **" "** \* 4  
  
 *# special symbols: +\*.| (in order of precedence highest to lowest, symbols coming before that  
  
 # union operator* bar\_pos = regex.find(**"|"**)  
 **if** bar\_pos != -1:  
 *# there is a bar in the string; union both sides  
 # (uses the leftmost bar if there are more than 1)* **return** get\_union(  
 get\_regex\_nfa(regex[:bar\_pos], indent),  
 get\_regex\_nfa(regex[bar\_pos + 1:], indent)  
 )  
  
 *# concatenation operator* dot\_pos = regex.find(**"."**)  
 **if** dot\_pos != -1:  
 *# there is a dot in the string; concatenate both sides  
 # (uses the leftmost dot if there are more than 1)* **return** get\_concat(  
 get\_regex\_nfa(regex[:dot\_pos], indent),  
 get\_regex\_nfa(regex[dot\_pos + 1:], indent)  
 )  
  
 *# kleene star operator* star\_pos = regex.find(**"\*"**)  
 **if** star\_pos != -1:  
 *# there is an asterisk in the string; wrap everything before it in a kleene star expression  
 # (uses the leftmost dot if there are more than 1)* star\_part = regex[:star\_pos]  
 trailing\_part = regex[star\_pos + 1:]  
 kleene\_nfa = get\_kleene\_star\_nfa(get\_regex\_nfa(star\_part, indent))  
  
 **if** len(trailing\_part) > 0:  
 **return** get\_concat(  
 kleene\_nfa,  
 get\_regex\_nfa(trailing\_part, indent)  
 )  
 **else**:  
 **return** kleene\_nfa  
  
 *# "one or more of" operator ('+' symbol)* plus\_pos = regex.find(**"+"**)  
 **if** plus\_pos != -1:  
 *# there is a plus in the string; wrap everything before it in the "one or more of" expression  
 # (uses the leftmost plus if there are more than 1)* plus\_part = regex[:plus\_pos]  
 trailing\_part = regex[plus\_pos + 1:]  
 plus\_nfa = get\_one\_or\_more\_of\_nfa(get\_regex\_nfa(plus\_part, indent))  
  
 **if** len(trailing\_part) > 0:  
 **return** get\_concat(  
 plus\_nfa,  
 get\_regex\_nfa(trailing\_part, indent)  
 )  
 **else**:  
 **return** plus\_nfa  
  
 *# "zero or one of" operator ('?' symbol)* qmark\_pos = regex.find(**"?"**)  
 **if** qmark\_pos != -1:  
 *# there is a question mark in the string; wrap everything before it in the "zero or one of" expression  
 # (uses the leftmost question mark if there are more than 1)* leading\_part = regex[:qmark\_pos]  
 trailing\_part = regex[qmark\_pos + 1:]  
 zero\_or\_one\_of\_nfa = get\_zero\_or\_one\_of\_nfa(get\_regex\_nfa(leading\_part, indent))  
  
 **if** len(trailing\_part) > 0:  
 **return** get\_concat(  
 zero\_or\_one\_of\_nfa,  
 get\_regex\_nfa(trailing\_part, indent)  
 )  
 **else**:  
 **return** zero\_or\_one\_of\_nfa  
  
 *# no special symbols left at this point* **if** len(regex) == 0:  
 *# base case: empty nfa for empty regex* **return** NFA()  
 **elif** len(regex) == 1:  
 *# base case: single symbol is directly turned into an NFA* **return** get\_single\_symbol\_regex(regex)  
 **else**:  
 *# multiple characters left; apply implicit concatenation between the first character  
 # and the remaining characters* **return** get\_concat(  
 get\_regex\_nfa(regex[0], indent),  
 get\_regex\_nfa(regex[1:], indent)  
 )

**OUTPUT:**



**Conclusion :** Successfully construct NFA using given regular expression.

**Reference:**

**https://www.geeksforgeeks.org/regular-expression-to-nfa/**

[**https://userpages.umbc.edu/~squire/cs451\_l7.html**](https://userpages.umbc.edu/~squire/cs451_l7.html)